NPS-CS-11-003

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Applying UML-based Formal Specification, Validation, and Verification to Space Flight Control System and Defense Software

by
Miriam C. Bergue Alves, Konstantin Beylin, Doron Drusinsky, James Bret Michael, and Man-Tak Shing

February 2011

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Executive Vice President and
Provost

The report was funded by the Graduate School of Operational and Information Sciences, Naval Postgraduate School (NPS).

This report was prepared by:

Doron Drusinsky
Associate Professor of Computer Science
Naval Postgraduate School

Man-Tak Shing
Associate Professor of Computer Science
Naval Postgraduate School

Dr. Miriam C.B. Alves
Research Associate
Contractual Support to NPS
Institute of Aeronautics and Space (IAE) at
DCTA (Brazilian Department of Aerospace
Science and Technology)

Chris Beylin
Software Engineer
Naval Air Warfare Center

James Bret Michael
Professor of Computer Science
Naval Postgraduate School

Reviewed by:

Peter J. Denning, Chairman
Department of Computer Science

Released by:

Karl A. van Bibber
Vice President and Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
1/31/2011	Technical Report	9/1/2010 – 1/31/2011

4. TITLE AND SUBTITLE
Applying UML-based Formal Specification, Validation, and Verification to Space Flight Software and Defense Software

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
M. C. B. Alves, K. Beylin, D. Drusinsky, J. B. Michael, M.T. Shing

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Naval Postgraduate School
1411 Cunningham Road, Monterey, CA 93943
and Naval Air Warfare Center, Weapons Division, Point Mugu, CA 93042

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-CS-11-003

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

This report presents the process and results of a formal computer-aided Specification, Validation and Verification (SV&V) of two mission and safety critical projects: the Brazilian Satellite Launcher flight software, and the Department of Defense's Multifunctional Information Distribution System (MIDS) controller. The Specification, Validation, and Verification (SV&V) process begins with a system requirement analysis and Natural Language (NL) specification. UML statechart-formal specification assertions are then created using the StateRover SV&V specification environment; these assertions formally capture the NL requirements. The assertions are validated against the NL and cognitive requirements using JUnit-based testing within the StateRover SV&V environment. Finally, Runtime Verification (RV) is performed on the target system under test (SUT). The RV phase is based on log files created by automatically instrumenting source code files, building and executing them on the VxWorks-based target thereby creating log files, importing resulting log files into the StateRover SV&V environment and executing them as JUnit tests against the assertions.

15. SUBJECT TERMS
specification, validation, verification, testing, flight software, UML, formal methods

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Doron Drusinsky
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UL		**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified			831-656-2168

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This report presents the process and results of a formal computer-aided Specification, Validation and Verification (SV&V) of two mission and safety critical projects: the Brazilian Satellite Launcher flight control system, and the Department of Defense's Multifunctional Information Distribution System (MIDS) controller. The Specification, Validation, and Verification (SV&V) process begins with a system requirement analysis and Natural Language (NL) specification. UML statechart-formal specification assertions are then created using the StateRover SV&V specification environment; these assertions formally capture the NL requirements. The assertions are validated against the NL and cognitive requirements using JUnit-based testing within the StateRover SV&V environment. Finally, Runtime Verification (RV) is performed on the target system under test (SUT). The RV phase is based on log files created by automatically instrumenting source code files, building and executing them on the VxWorks-based target thereby creating log files, importing resulting log files into the StateRover SV&V environment and executing them as JUnit tests against the assertions.

THIS PAGE INTENTIONALLY LEFT BLANK

1. INTRODUCTION

Correctness of computer software is critical, especially for embedded software in mission critical application domains such as space, transportation, military and medical. These software systems perform the mission and safety-critical functions with complex time-constrained sequencing behaviors that are difficult to specify correctly and to verify using the prevailing manual SV&V techniques. Research has shown that formal specifications and formal methods help improve the clarity and precision of requirements specifications [1]. In particular, lightweight formal methods, when used in tandem with rapid prototyping, help debug the requirements and identify errors earlier in the design process [2]. In [3], we presented a continuous and proactive process for conducting V&V of systems. The process involves using scenario-based testing to validate whether the formal assertions correctly capture the intent of the natural language requirements, and is automated through the use of statechart-assertions and runtime execution monitoring [4].

This paper presents the results of applying the method to the SV&V of two mission critical, time-constrained systems: (i) the Satellite Launcher flight control system [5], and (ii) the Multifunctional Information Distribution System (MIDS) controller. This is the first reported case of the application of the entire SV&V methodology using rapidly developed UML-based formal specifications. It is also the first reported case of "distributed verification", where specification and validation of the part of the flight control system was performed in California, execution and log file creation was performed in Brazil, and subsequent log-file based RV was performed in California.

2. PRELIMINARIES - THE V&V PROCESS

The SV&V process consisted of the following activities:

1. Perform system requirement analysis and Natural Language (NL) specification in California.
2. Create UML statechart-formal specification assertions using the StateRover SV&V specification environment [6]; these assertions formally capture the NL requirements. The assertions are then validated against the NL requirements using JUnit-based testing within the StateRover SV&V environment. This activity was performed in California.
3. Finally, Runtime Verification (RV) is performed on the target System Under Test (SUT). The RV phase was based on log files, as follows:
   a. StateRover is used to automatically instrument the source code files; this activity was performed in California.
   b. The instrumented version of the software is built for the VxWorks-based target; in the case of the Brazilian Satellite Launcher flight control system, this activity was performed in Brazil .

c. The resulting program was executed on the VxWorks-based target thereby creating log files; in the case of the Brazilian Satellite Launcher flight control system, this activity was performed in Brazil.

d. The log files are imported into the StateRover SV&V environment using the StateRover "Import" tool that converts log files into an equivalent JUnit test. Such JUnit tests are called *JUnit verification tests*, thereby distinguishing them from validation tests discussed in activity No. 2. This activity was performed in California.

e. A namespace mapping is created, using the StateRover namespace mapping tool. This mapping translates the C functions and variables namespace to the namespace used in the assertions. This activity was performed in California.

f. The JUnit verification tests are executed against the assertions, using the StateRover's assertion repository tool, a plug-in that dispatches events (corresponding to function calls or variables assignments logged in the log file) to all assertions. This activity was performed in California.

Requirements validation, as established in activity No. 2, follows a abstract-validate-refine strategy (Figure 1) and involves the following steps:

1. Requirements Modeling: given a set of natural language requirements, statechart-assertions are created. They represent the understanding of the cognitive requirements model.
2. Validation: for every requirement, a set of scenarios is created to check whether the assertion satisfies the desired system behavior as specified in the NL requirements. Some scenarios will satisfy, or conform the requirement, and other scenarios will fail it.

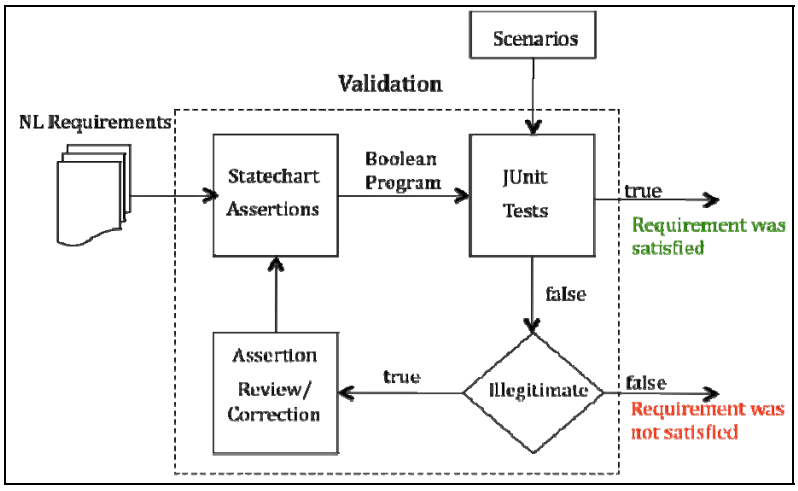


Figure 1. The validation activities.

- Failure Scenarios: the scenario that leads the assertion to fail is examined to determine whether it is illegitimate. This examination is done by evaluating the

requirement, using the failure scenario as a guide, to find out if the scenario represents a valid behavior of the system. If this is the case, the problem is reported (requirement was not validated). Otherwise, the loop proceeds to the next step.

- Assertion Refinement: the set of statechart assertions is changed to eliminate the wrong behavior and possibly other illegitimate behaviors introduced in the abstraction process. Given the update set of statechart assertions, the loop proceeds to step 1.

Figure 2 describes the verification process framework based on the abstract-verify-refine strategy. It uses the following steps to gather data produced by the runtime execution in the log files and then verifies the test results via automated generated JUnit tests based on these log files.

1. Runtime execution monitoring: data is gathered in a log file by observing the system behavior in real time execution.
2. Log-file based Verification: a set of JUnit verification tests is created from the log-files. They are executed against the assertions; some scenarios will satisfy, or conform the corresponding requirements, while other scenarios will not.
3. Failure tests: verification tests that fail an assertion are examined to determine whether they are due to an implementation error or perhaps the log files were generate under abnormal operational conditions. In the latter case, the problem is reported. Otherwise the process proceeds.
4. Assertion Refinement: feedback is given to the system design and implementation team. Given the update design and code, the loop proceeds to step 1.

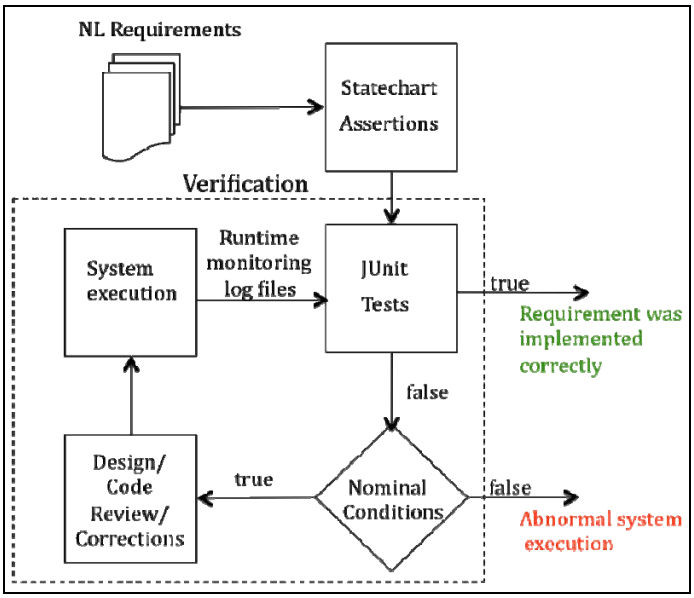


Figure 2. The verification activities

3. FLIGHT CONTROL SYSTEMFORMAL SPECIFICATION

The SUT has algorithms and control loops that operate during the powered and ballistics flight phases to direct the actuators, so as to keep the launcher sufficiently close to its reference launching trajectory. The sequence of flight events characterizes the different stages of the flight and determines when the algorithms and control loops have to be executed during the flight [7]. There are two different types of events: reference events and relative events.

The flight control system, in a pre-determined timeframe, must detect the four reference events:

- Reference event named *A*: LiftOff.
- Reference event named *B*: ThrustDrop_1Stage.
- Reference event named *C:* ThrustDrop_2Stage.
- Reference event named *D*: ThrustDrop_3Stage.

Once a reference event is detected, the flight control system has to detect relative events in a pre-determined time in order to activate and/or deactivate controlling algorithms and electrical systems (e.g. actuators, ignition devices, stage separation dispositive).

Figure 3 shows the statechart-assertion formalization for the reference event *A* requirement:

> *"Req_Ref_A: Once the navigation starts (time=0), A must occur within the interval [lA,uA]."*

The statechart-assertion in Figure 3 will enter the *Error_A* state if it observes the event *A* when in state *Nav_On*, before its lower limit time (timer_LA=) or when in state *Waiting_A*, it observes that the upper limit of the time interval (timer_UA) has been reached and *A* was not detected.

StateRover's code generator generates a Java class *Req_Ref_A* for the statechart-assertion file. The generated code is designed to work with the JUnit Java testing framework [8].

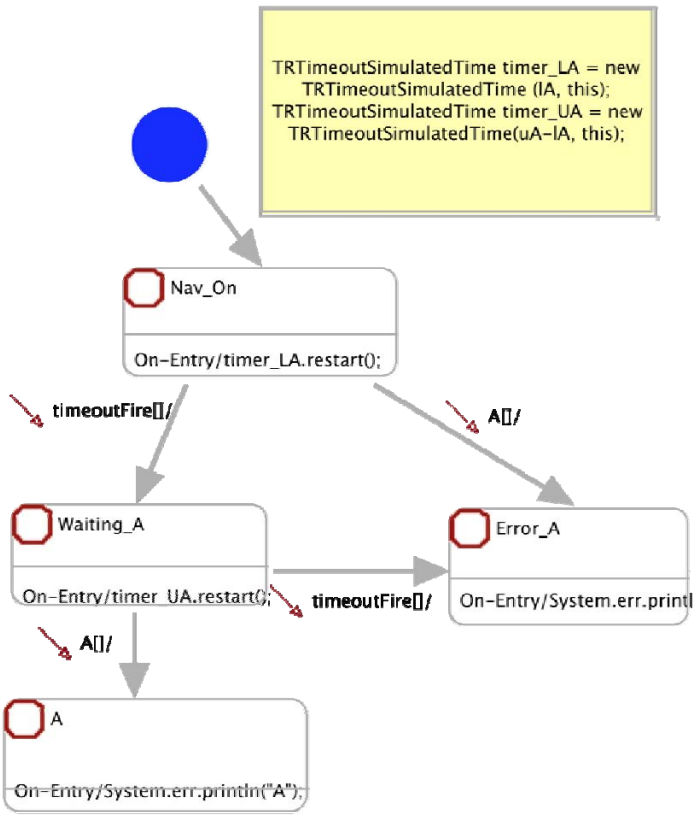


Figure 3. Statechart-assertion for the *Req_Ref_A*

Once the reference event *A* is detected, eleven other relative events associated with the event *A* must be detected after a time interval pre-established for each relative event, with X milliseconds of tolerance. This tolerance tends to be readjusted for each assertion as much as we learn about the system performance. Let us generically consider the relative event *Ax*, and its detection time *DAx*, x=0,1,..10. Therefore, the relative event requirement is written as:

> *"Req_Rel_Ax: If A occurs, then Ax must occur DAx millisecs within X millisec afterwards."*

A single statechart-assertion formalization for these requirements was used as a pattern, as shown in Figure 4. The associated Java class generated by StateRover was then re-factored for each of the relative requirements and a special Java function was created to set the corresponding *DAx* for each *Ax*.

The first part of the statechart-assertion in Figure 4 formalizes the detection of the event *A* (If A is detected), as explained previously. The remaining part formalizes the generic relative event detection (*R_E*) after *A* has been detected (then Ax must be detected *DAx* millisecs, within X millisecs afterwards). The statechart-assertion will enter the *Error_Rel* state if it observes the *R_E* event when in state A, before its DAx milliseconds timeout (*timer2*) has expired or when in state *Waiting_Relative* for more than X milliseconds (*timer3*).

The statechart-assertion for the reference event *B* requirement is presented in Figure 5:

"Req_Ref_B: B must be detected within interval [lB,uB] of the detection of A."

The statechart-assertion in Figure 5 will enter the *Error_B* state if it observes the *B* event when in state *A*, before its lower limit time (*timer_LB*) or when in state *Waiting_B*, it observes that the upper limit of the time interval (*timer_UB*) has been reached and *B* was not detected.

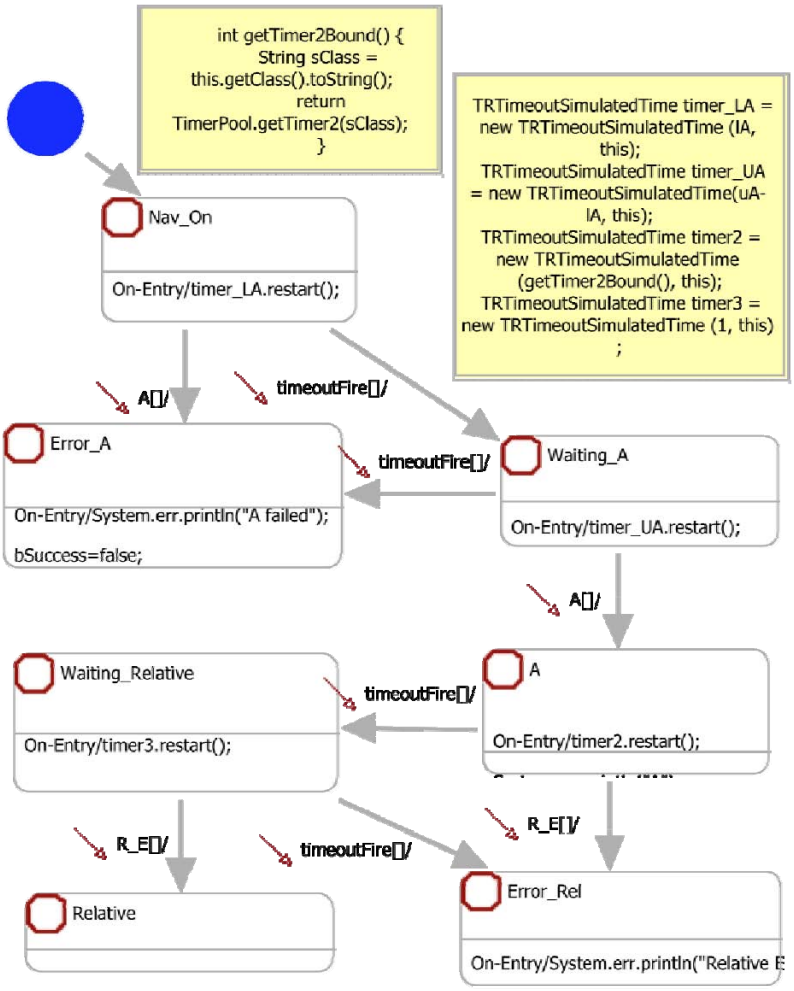


Figure 4. Statechart-assertion for the Req_Rel_Ax

The requirements for the other reference events C and D were established as follow:

"Req_Ref_C: C must be detected within interval [lC,uC] of the detection of A."

"Req_Ref_D: D must be detected within interval [lD,uD] of the detection of C."

The statechart-assertion for the requirements *Req_Ref_C* uses the same pattern showed in Figure 5 and the statechart-assertion for *Req_Ref_D* is illustrated in Figure 6.

One can observe that this requirement has a similar formalization as the reference event B, but D also depends on C.

The relative events requirements associated with the reference events B, C and D follows the same pattern as the ones associated with reference event A, as showed previously in Figure 4.

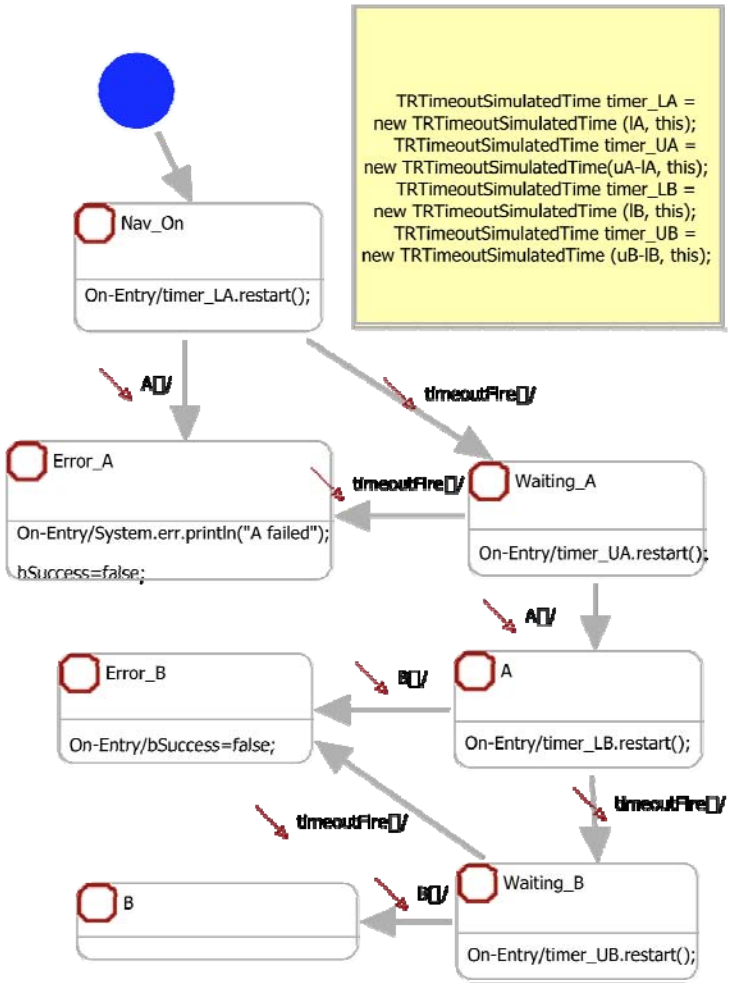


Figure 5. Statechart-assertion for the *Req_Ref_B*

Up to this point, 44 requirements associated with the flight events sequence, including reference and relative events, were formalized as statechart-assertions. The support of the StateRover tool and the reuse of statechart-assertions were fundamental to have the work accomplished, as they both contributed to the rapid creation of assertions.

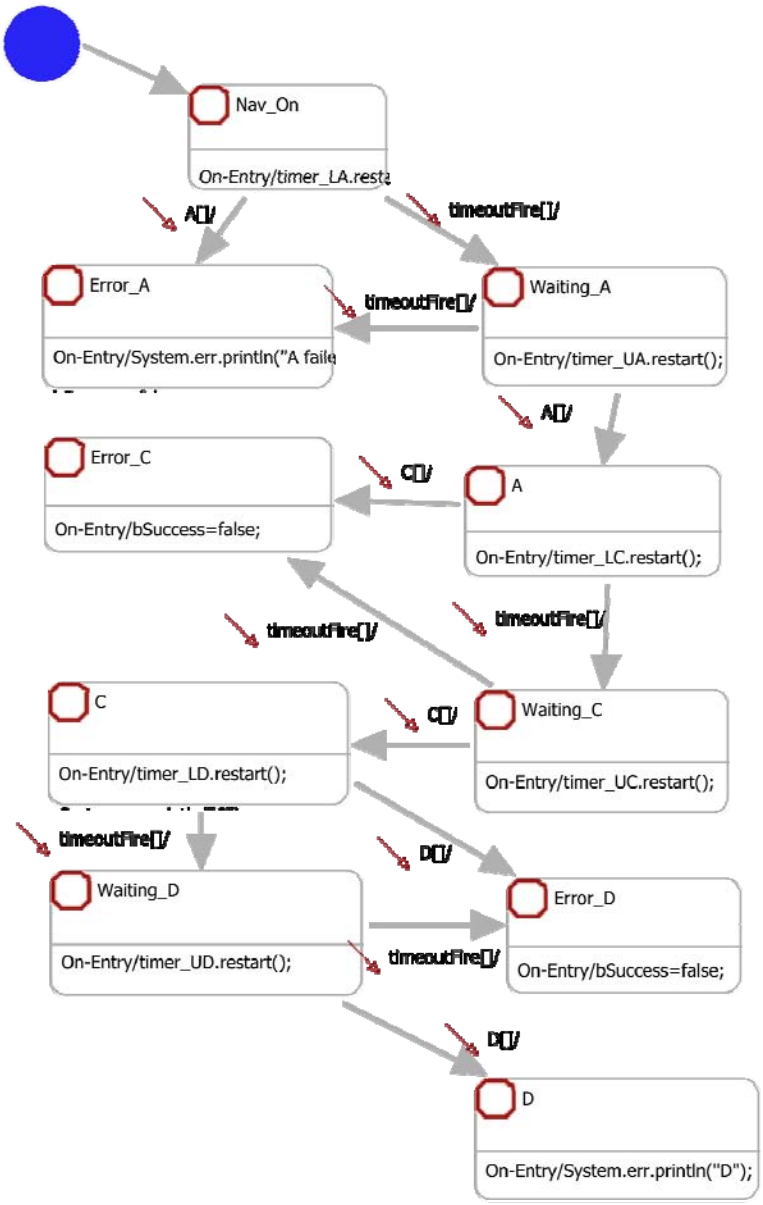


Figure 6. Statechart-assertion for the *Req_Ref_D*

4. FLIGHT CONTROL SYSTEM VALIDATION

To assure that the statechart assertions accurately represent the requirements, their behaviors were tested using JUnit test cases created as follows:

- Obvious success: test a trivial scenario that conforms to the NL requirement;
- Obvious failure: test a trivial scenario that violates the NL requirement.

- Full scenario success: test a nontrivial scenario that goes through the entire basic scenario while in agreement with the NL requirement.
- Full scenario failure: test a nontrivial scenario that goes through the entire basic scenario while violating the NL requirement.

When necessary, additional JUnit tests were created to guarantee that all the states in the state-assertion diagram were covered.

Listing 1 shows the JUnit test case *A0_test1* for the relative event *A0* requirement (statechart-assertion in Figure 5), and Figure 7 shows the respective timeline associated with this test. This test represents a full scenario success, where *A0* is supposed to occur 15 milliseconds after *A* occurs.

```
package req_ref_timer_A0;
import junit.framework.TestCase;
public class A0_test1 extends TestCase {
    req_rel_A0 A0;
    protected void setUp() throws Exception {
         super.setUp();
         A0=new req_rel_A0();
    }
    protected void tearDown() throws Exception {
         super.tearDown();
    }
    public void testExecTRreset(){
         A0.incrTime(5000);
         A0.A();
         A0.incrTime(15);
         A0.R_E();
         A0.incrTime(1);
         assertTrue(A0.isSuccess());
    }
}
```

Listing 1. JUnit test case *A0_test1*

Figure 8 shows two timelines associated with two failure scenario tests for the relative event *A0* requirement: the first one is a full scenario failure where the event *R_E (A0)* occurred too late (*A0_test2*); the second one shows a sequence of events where *R_E* occurred too early (*A0_test3*).

StateRover for Eclipse animation was used to visualize the behavior of the statechart-assertions while running the validation tests. It also helped to analyze the statechart-assertion state coverage. Figure 9 presents the statechart-assertion view when the animation of *A0_test2* was done. The last two visited states of the statechart-assertion are highlighted.

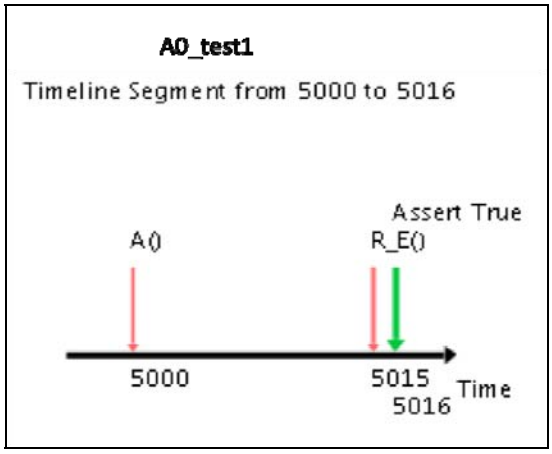


Figure 7. A full scenario success sequence of events

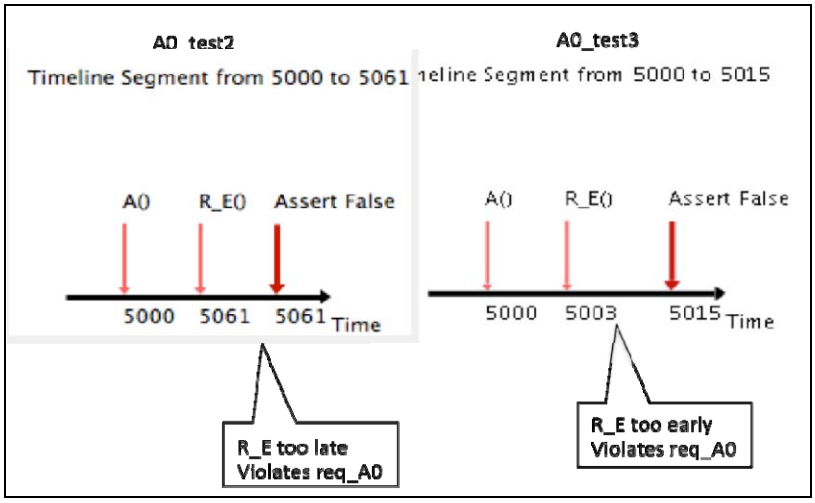


Figure 8. Two event sequences that violates requirement A0

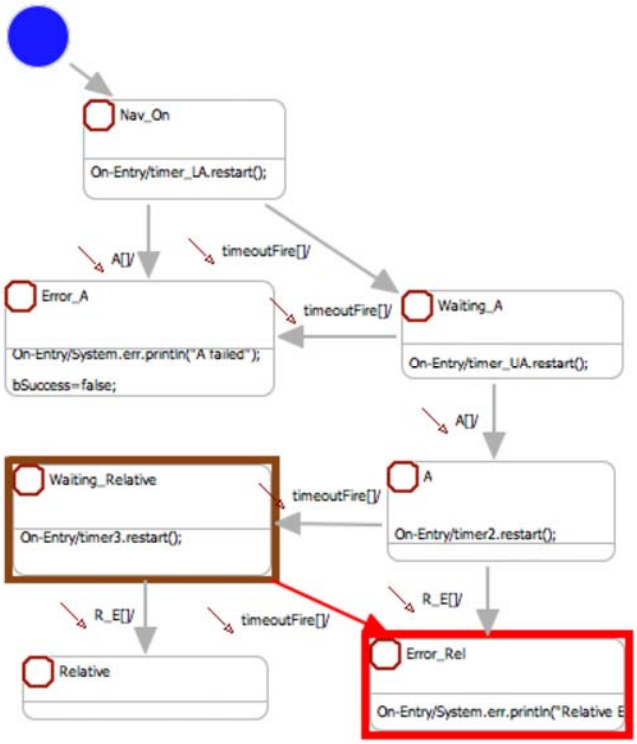


Figure 9. Statechart-assertion animation for a full scenario failure.

As of January 2011, around 220 JUnits validation tests were run to validate 44 requirements.

5. FLIGHT CONTROL SYSTEMVERIFICATION

A simplified scheme of the laboratory environment where the runtime execution monitoring took place is presented in Figure 10. The flight control system is embedded in a target computer running a specific RTOS (VxWorks) connected to a Host computer where the log files are created.

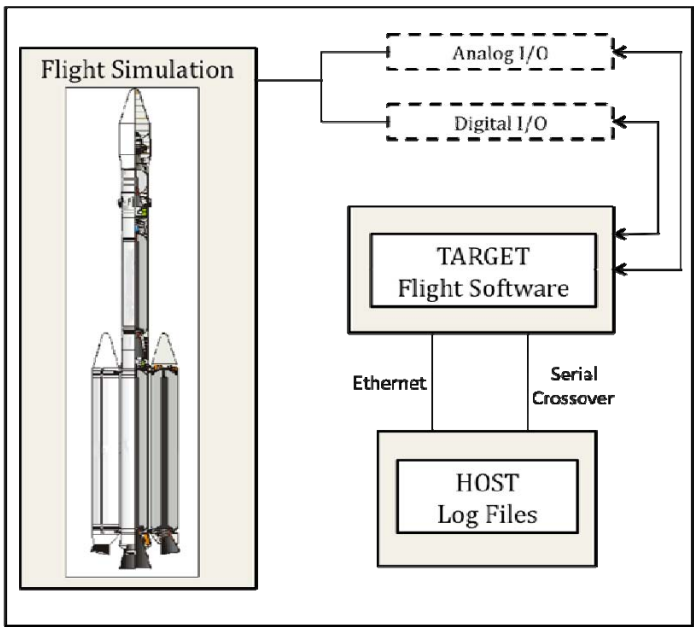


Figure 10. The lab setup scheme for runtime execution monitoring

In the specific case of this study, adaptations in the automatic instrumentation had to be done in order to optimize the execution of the instrumented code. This was necessary due to the very tight time constraints imposed to the flight control system execution. In the first simulations, the overhead of getting extra data interfered in the results of the simulated flight. Therefore, changes were made in the instrumented code to cope with the restrictions imposed by the flight control system environment.

As the initial main goal was monitoring the flight events sequence, the code instrumentation was optimized to collect certain variables assignments that flagged the events occurrence and their current time of occurrence.

While still working in the generation of additional log files, we have created three log files to date. It is worth mentioning that because log files generation took place in Brazil, their throughput was subject to the availability of the Flight Dynamics Lab [9][10]. In order to generate JUnit verification tests to verify the statechart-assertions, the log file was imported by StateRover, which generates a XLM equivalent file and a JUnitFromLogs Java class. This class contained the log file-based verification tests for the statechart-assertions.

The next step, prior to running the verification tests, was to create a namespace mapping that mapped the SUT name space to the assertion namespace. Figure 11 shows a namespace mapping for one of the log files.

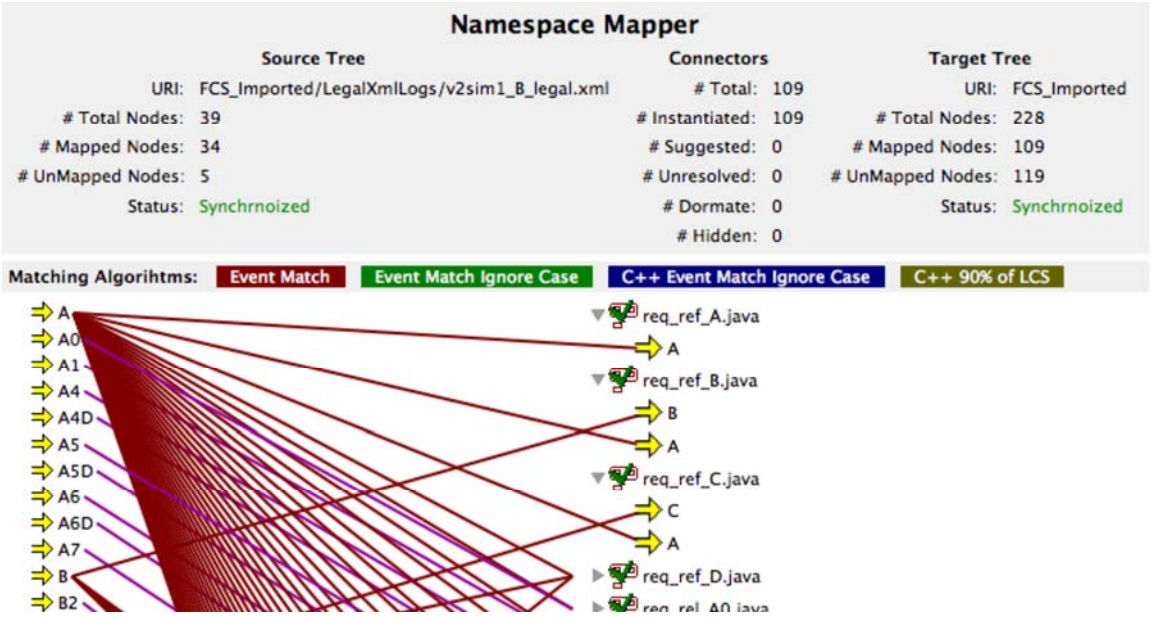


Figure 11. Namespace mapping for verification testing

The left hand side of the Namespace Mapper shows nodes consisting of events from the log file; The right hand side of the Namespace Mapper shows nodes consisting of events and conditions for assertions in the assertion repository, namely statechart assertions and propositional assertions (outside the scope of this report. The connectors connecting both sides can be created manually or algorithmically – using built in or custom algorithms. In this study some of the connectors were created automatically and others were created manually, using a drag-and-drop user interface.

The verification tests were executed according to the scheme shown in Figure 12. It could be observed that approximately 50% of the assertions were violated due to late reference events detection. The reason behind this high number is attributed to the execution overhead caused by the tight timing on the target that was perturbed by the instrumented code causing a cascade effect that results in the late detection of the relative events as well. This situation was rather expected, considering the strict real time requirements of the flight control system and its operation environment.

However, the assertions violations in certain scenarios uncover the lack of well-defined requirements to deal with recovery of failures and adequate treatment for missing time deadlines. Table 1 summarizes the tests and their results for the V&V process. The results also showed that the adopted process was possible due to the support of a computer-aided tool, and more efficient when compared to traditional and manual techniques. It significantly improved the requirement understanding, validation and verification.

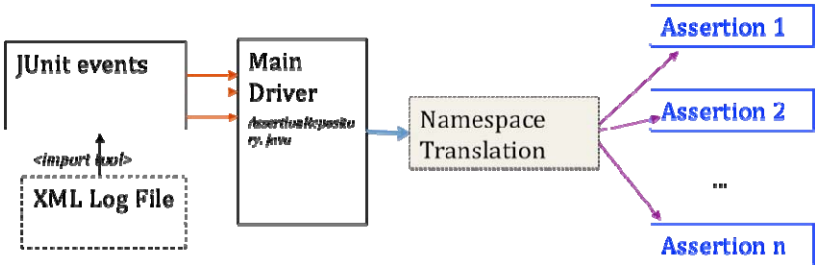


Figure 12. Verification tests execution from log files

Table 1. Summary of the Validation and Verification tests.

	Validation Tests (% of assertions)	**Verification Tests (% of assertions)**
Success	60%	50%
Violated	40% (*)	50%
Total of tests	220 (5 tests per assertion) 220 JUnit classes - 1 JUnit class per test	3 log files (3 tests per assertion) 3 JUnit class- 1 JUnit class per log file

(*) obvious failure and full scenario failure

6. THE MIDS CONTROLLER SYSTEM UNDER TEST (SUT)

The EA-6B mission supports Suppression of Enemy Air Defenses (SEAD) and Destruction of Enemy Air Defenses (DEAD). The mission includes detecting, locating, identifying, correlating on-board and off-board data, and employing jamming techniques against enemy communication and weapon systems. It also includes employing or directing the employment of weapons to enemy assets. The modern battle space is complex and dynamic, requiring timely and clear information and decisions by all levels of military command. Link-16 supports these constraints by enabling exchange of real-time tactical data among US Navy, Joint Service, and North Atlantic Treaty Organization (NATO) ships and aircraft. Link-16 provides for the rapid and reliable exchange of tactical data at all levels of command, control and operational engagement. It consists of a specialized communications network infrastructure operating in the UHF part of the Radio Frequency (RF) spectrum. The Multifunctional Information Distribution System (MIDS) is a hardware communication device that enables Link-16 data and voice communication and access to the Link-16 network. It implements Link-16 tactical communication by providing integrated position determination, navigation and present position identification as well as voice and data communication capabilities. Any system requiring Link-16 network capabilities has to interface with a MIDS terminal.

On the EA-6B aircraft, MIDS Controller (MC) is a designated host computer to interface with a MIDS terminal.

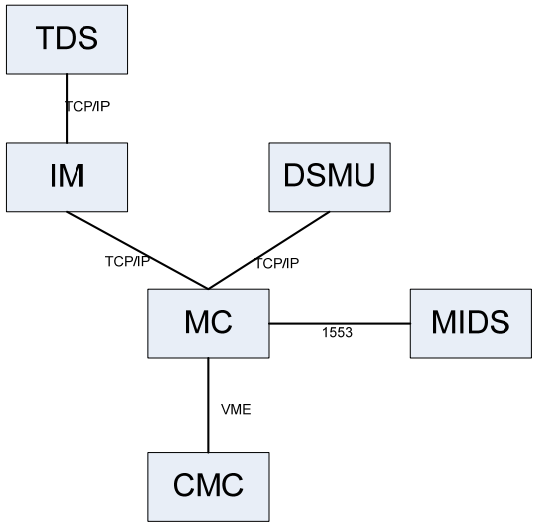


Figure 13. MC components and connections

6.1 SYSTEM ARCHITECTURE

MC is developed in C++ with about 60K source line of code. MC is hosted on PowerPC single board computer running VxWorks real time operating system. The system architecture from the MIDS Controller (MC) perspective includes the components and connections depicted in Figure 13. The MC components are:

- Tactical Display System (TDS): Provides operator controls and displays, Link-16 tracks management and MIDS terminal control.
- Information Manager (IM): Manages distributed databases, coordinates Link-16 data between MC and TDS, distributes data to multiple TDSs, mission recording and mission database loading and storage.
- Data Storage Memory Unit (DSMU): Stores operational programs, mission database and mission recording data.
- Multifunctional Information Distribution System (MIDS): Link-16 network participation, receive and transmit Link-16 messages.
- Central Mission Computer (CMC): RF jamming management and control of the RF synthesizers and transmitters. Aircraft navigation data control.
- MIDS Controller (MC): Intermediary between MIDS terminal and EA-6B integrated distributed systems. Provides data exchange protocol with a MIDS terminal to process Link-16 messages. Implements Link-16 message formatting and message decoding. Maintains Link-16 all active tracks, local and remote tracks management and track filtering.

6.2 LINK-16 MESSAGES

The messages exchanged between participating Link-16 platforms are called J-series messages. Each J-series message is composed of one or more words. All messages have an initial word (I). In addition, they may have one or more extension words (E), and one or more continuation words (C). Each J-series message is mapped to the functional virtual circuit (surveillance, air control, Electronic Warfare (EW), etc.) it supports. These virtual circuits are called Network Participation Group (NPGs) and are the building blocks of the network. Various types of operational information are exchanged among users through assigned access to these NPGs. MC categorizes incoming J-series messages into two broad categories:

1. Situational Awareness (SA) messages providing data regarding Track/Point entities.
2. Command messages directing/requesting either the MC or the aircrew to act. Alerts and warnings are included in this category. Some of these messages will be Link-16 Receipt Compliance (R/C) messages requiring an Operator Response (OR) message.

6.3 NL REQUIREMENTS

The NL requirements were taken directly from MC Software Requirement Specification (SRS) document. The NL requirements described in this paper capture two distinct functional areas: MC Power Up Initialization and HARM (High Speed Anti-Radar Missile) Hand Off (HHO).

6.3.1 MC Power Up Initialization Sequence

Note that in the NL specifications below, numbered actions must be performed in sequence while subsidiary actions may be performed in any order.

1. The MC shall obtain results of the PowerPC power-on Built-in Test (BIT)
2. The MC will perform the following actions (order is irrelevant):
    a. The MC shall begin transmitting its UDP Broadcast Client Status message-2541
    b. The MC shall attempt to establish MIL-STD-1553B communications with the MIDS LVT and shall continue to do so until communications are established.
    c. When communications are established, MC shall obtain the results of the terminal's power-on BIT
    d. When communications are established, MC shall set the LVT Terminal State (AP004) to Time Division Multiple Access (TDMA) Only
    e. When communications are established, MC shall obtain the terminal's current load and net entry states
    f. The MC shall attempt to establish a TCP connection with the IM processor
    g. The MC shall set up an VME bus interrupt capability to receive an external interrupt from CMC.
3. Once the TCP socket is established, MC will send the following messages to the TDS (order is irrelevant):
    a. MC shall send the On-Demand BIT (ODB)/Power-On BIT Results message-2544, containing results of PowerPC and MIDS LVT power-on BIT.
    b. MC shall send the MC Software Version message-2547.
    c. MC shall send the Link-16 Network Status message-2545.
4. In addition, MC shall complete entire initialization sequence within 10 seconds.

6.3.2 HARM Hand Off (HHO)

High Speed Anti-Radar Missile is used against detected radar sites. Link-16 is used to coordinate HARM missile launch between platform with surveillance capabilities and HARM shooters. For example, EA-6B has a sophisticated surveillance system, but might not have HARM on board. F-16 has HARM launching capabilities but does not have a sophisticated surveillance detection system. In this case, Link-16 is using a series of J-Messages J3.5 (radar location), J12.6 (HARM parameters) and J12.0 (actual command to shoot HARM missile) to coordinate HARM shot between EA-6B and F-16.

HHO Requirements include:

- Upon receiving from IM message-2572 (Local Track Report) with HHO field set to true, MC shall: create local track in the database, assign loopbackID to a track number, setup retry count to one and transmit J3.5 (Land Point) message to MIDS terminal.

- MC shall receive loopbackID response message from the MIDS terminal for the J3.5 message. If loopbackID from MIDS indicating transmit status failure, MC shall re-transmit J3.5 message once. If loopbackID from MIDS indicating transmit status failure from the re-transmit J3.5 message, MC shall abort HHO processing.
- Upon receiving loopbackID from MIDS indicating transmit success for the J3.5 message, MC shall assign loopbackID to a track number, setup retry count to one and transmit J12.6 (HARM DA parameters) message to MIDS terminal.
- MC shall receive loopbackID response message from the MIDS terminal for the J12.6 message. If loopbackID from MIDS indicating transmit status failure, MC shall re-transmit J12.6 message once. If loopbackID from MIDS indicating transmit status failure from re-transmit J12.6 message, MC shall abort HHO processing.
- Upon receiving loopbackID from MIDS indicating transmit success for the J12.6 message, MC shall wait for the IM to send to MC Message-5362 (Mission Assignment).
- Upon receiving from IM message-5362 (Mission Assignment), and only if MC had already successfully processed J3.5 and J12.6, MC shall assign loopback to a track number, setup retry count to one and transmit J12.0 Mission Assignment.
- MC shall process message-5362 from IM only if MC had successfully sent J3.5 and J12.6 messages.
- In case IM sends Message-5362 prior to Message-2572, MC must wait for Message-2572 from IM, in order to transmit J3.5 (Land Point) and J12.6 (HARM DA parameters) messages prior to transmit J12.0 (Mission Assignment).
- MC shall receive loopbackID response message from the MIDS terminal for the J12.0 message. If loopbackID from MIDS indicating transmit status failure, MC shall re-transmit J12.0 message once. If loopbackID from MIDS indicating transmit status failure from re-transmit J12.0 message, MC shall abort HHO processing.
- Upon receiving loopbackID from MIDS indicating transmit success for the J12.0 message, MC shall wait for the J12.0 message from the MIDS terminal with a Receipt Compliance fields indicating the following response: either Will Comply or Cannot Comply status.
- Upon receiving J12.0 message from MIDS terminal with Cannot Comply response, MC shall remove local track from the database.

7. MC FORMAL SPECIFICATION AND VALIDATION

Figure 14 depicts the statechart assertion diagram for the MC Power Up Initialization Sequence described in section 6.4.1. It captures the following NL concerns:

1. The NL requirement specifies a strict sequence order for sub-requirements 1, 2, and 3.
2. There exists a 10 seconds upper bound time constraint for the entire initialization sequence.

3. The order of events within numbered NL requirements 2(BIT) and 3(Interfaces Initialization) is irrelevant, except for MC having to first establish 1553 communication with a MIDS terminal and only then get MIDS terminal BIT, set TDMA and get terminal load and net status.

Figure 15 depicts the statechart assertion diagram for the HHO requirement set described in section 6.4.1.

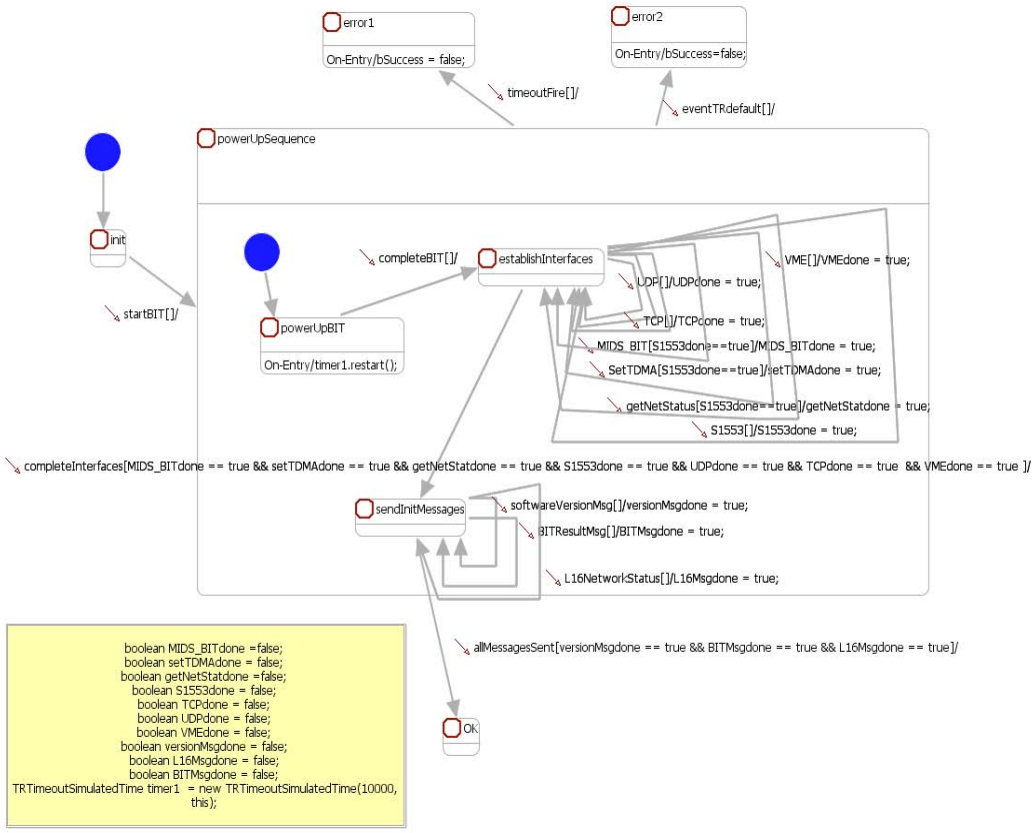


Figure 14. Statechart assertion for NL requirement 6.4.1.1

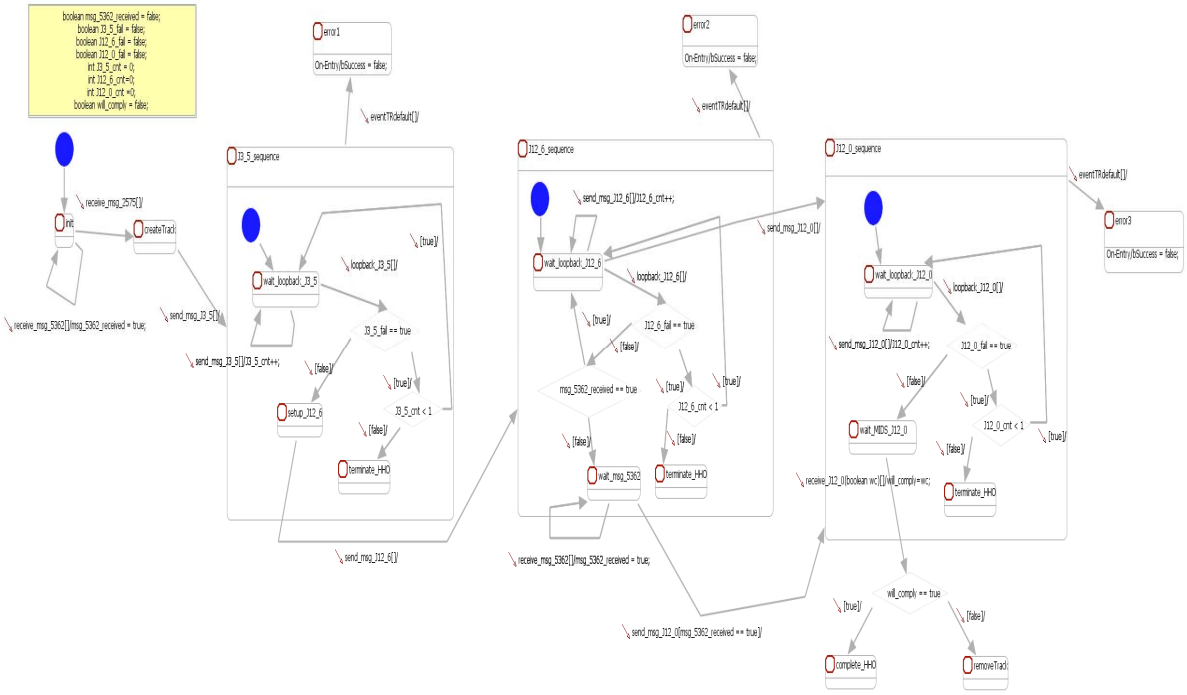
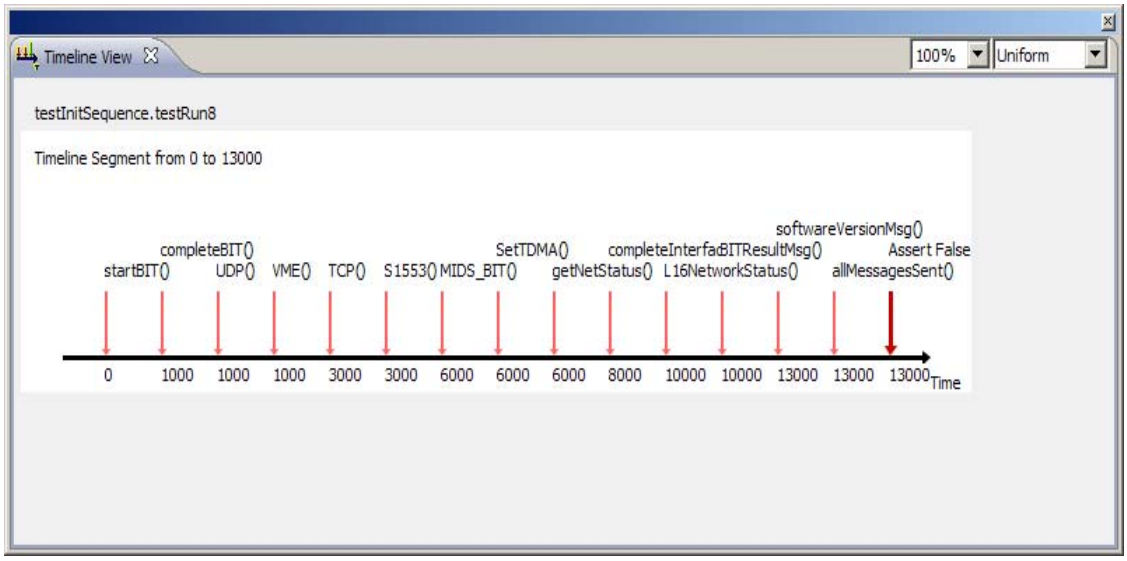


Figure 15. Statechart assertion for NL requirement 6.4.1.2

Based on the validation methodology described in [5], we created a plurality of JUnit validation tests for each statechart assertion. Two such tests, for the MC Power Up Initialization statechart assertion, are depicted using the timeline diagrams of Figure 16. A timeline diagram captures events and their associated time stamp of occurrence. The timeline diagrams of Figures 16a and 16b depict success and failure cases, respectively. Specifically, Figure 16a depicts a test for a scenario the developer believes conforms to the NL whereas Figure 16b depicts a test for a scenario the developer believes violates to the NL. Additional validation tests were created according to the validation testing patterns methodology of [5].

a. Timeline diagram rendering of a validation test expecting a success.



b. Timeline diagram rendering of a validation test expecting a failure (i.e., violation of a NL requirement).

Figure 16. Two validation tests for the MC Power Up Initialization statechart assertion.

The statechart assertions were developed and validated inside a special container called the assertion repository. This is a special Eclipse project that provides the

environment for subsequent verification described in section 8. Figure 17 depicts the Eclipse view of the assertion repository for the MC controller.

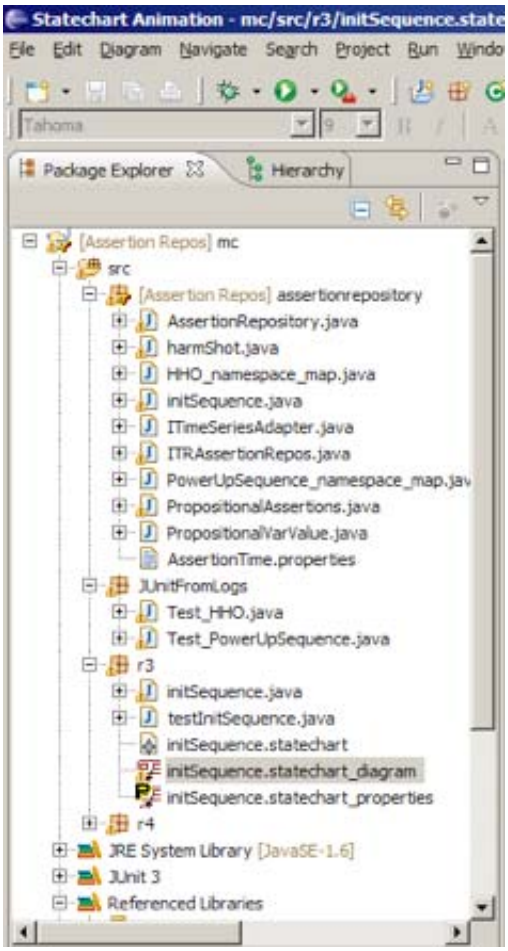


Figure 17. The assertion repository view for MC

8. VERIFICATION USING SOURCE CODE INSTRUMENTATION AND LOG FILES

Verification was performed on the VxWorks-based target system using the steps similar to those presented in section 5.

8.1 Source Code Instrumentation

Using an instrumentation GUI within the StateRover we selected the project and individual files for instrumentation. The source code instrumentation tool then automatically inserted instrumentation code snippets into the source file, resulting in instrumented code as in Listing 2.

Note that although the instrumentation tool, like all other StateRover tools, is an Eclipse plug-in, the source code was not developed and is not required to be built in the Eclipse environment. The MC source code was developed using VxWorks development environment and is built there. The source code files were imported into Eclipse solely for the purpose of instrumentation. They were subsequently used to rebuild the project in the VxWorks environment.

```
void TerminalCtrl::periodicTask()
{
   int               iCycleCount   = 0;
   int               cmcCycleCount = 0;
   unsigned int      mstrRadValidFlags01;
   unsigned int      mstrRadValidFlags02;
   MHCLANMsg         *mhcLANMsg, *mstrRadLANMsg;
   MHCLANMsg         *rpLAN;
   STATUS            result = OK;

   /*  @instrumented Maya-Software. */
   MAYA_INSTRUMENT((char *)
     "<sig><![CDATA[TerminalCtrl::periodicTask()]]></sig>",
     (char *)"",(char *)
       "<sourcefile><![CDATA[C:/mcWorkSpace/verificationMC/
        terminalctrl.cpp]]></sourcefile>",
     (char *)"<pos val=\"47965\" instrval=\"54507\" />");

   pCMC = new  CMCDATAMsg;
   CMCDATAMsg *      airStatus_CMC = new CMCDATAMsg;
   airStatus_CMC->id  = CMCIN_AIRCRAFT_STATUS;
   CMCin_AircraftStatus airStatus;

   d1553StatusInfo_t statusInfo;

   int tickDelayBeforeStart =
     sysClkRateGet() / 20 - TC_DELAY_TICK;
   periodicTaskStarted = true;
```

Listing 2. A source code file snippet with source code instrumentation.

The snippet is automatically inserted to the source code file using the instrumentation tool.

The instrumentation snippet makes calls to customizable utility methods that log the following information in a log file. The information being logged is:

- Per every method (or C function) call:
  - Method signature and actual arguments.

- Method call time.
- Location of instrumentation point in source code file (optional).

- Per variable assignments: name and value of variable being assigned.

8.2 Log file Creation

After instrumentation, the MC application was re-built in the VxWorks environment and then executed on the target using the existing test harness. A resulting log file snippet (one of a plurality of log files created by a plurality of test runs) is depicted in Listing 3.

```
<newtest>
<event>
<sig><![CDATA[TerminalCtrl::snd1553_BIMA01()]]></sig>
<time lang="c" unit="sec" val="430" />
</event>
<event>
<sig><![CDATA[TerminalCtrl::snd1553_BIMA04()]]></sig>
<time lang="c" unit="sec" val="430" />
</event>
<event>
<sig><![CDATA[CMCInterface::getReadDataReadyFlag(
   map<unsigned short,
            cmc_data_info>::iterator& pos )]]></sig>
<time lang="c" unit="sec" val="430" />
</event>
<event>
<sig><![CDATA[TerminalCtrl::get1553_BOMA01()]]></sig>
<time lang="c" unit="sec" val="430" />
</event>
<event>
<sig><![CDATA[TerminalCtrl::get1553_BOMA08()]]></sig>
<time lang="c" unit="sec" val="430" />
</event>
```

Listing 3. A log file snippet

8.3 Log File Import and JUnit Conversion

Log-files were imported into the assertion repository using the StateRover's import plug-in. This tool converts the log file from XML format to an equivalent verification JUnit test. Listing 4 contains a snippet of the verification JUnit test that corresponds to the log file of Listing 3.

```
public void testMe1() throws Exception {
   assertions.reset(
     "assertionrepository.HHO_namespace_map");
```

```
   nBaseTime = (int)(430L  * nFactor_LogUnitsAreSec);
   assertions.fire("TerminalCtrl::snd1553_BIMA01()");
   assertions.incrTime(
     (int)(430L  * nFactor_LogUnitsAreSec - nBaseTime));
   nBaseTime = (int)(430L  * nFactor_LogUnitsAreSec);
   assertions.fire("TerminalCtrl::snd1553_BIMA04()");
   assertions.incrTime(
     (int)(430L  * nFactor_LogUnitsAreSec –
           nBaseTime));
   nBaseTime = (int)(430L  * nFactor_LogUnitsAreSec);
   …
}
```

Listing 4. A snippet of the verification JUnit test created from the log file depicted in Listing 2.

8.4 Namespace Mapping

The next step in the MC verification process was to match the namespace used by the assertions to the C++ namespace of the MC code base, namely, to the namespace of the method calls logged in the log file. This mapping is done using a namespace mapping GUI, part of the StateRover's namespace plug-in. This GUI allows manual and algorithmic mapping (using built-in as well as customizable mapping algorithms) of the two namespaces. An instance namespace map used for MC verification is depicted in Figure 18. For example, it maps ProcessLoopbackResponse to Loopback_J12_0. The namespace mapping plug-in also generates Java code (denoted executable namespace translation) that implements this mapping.

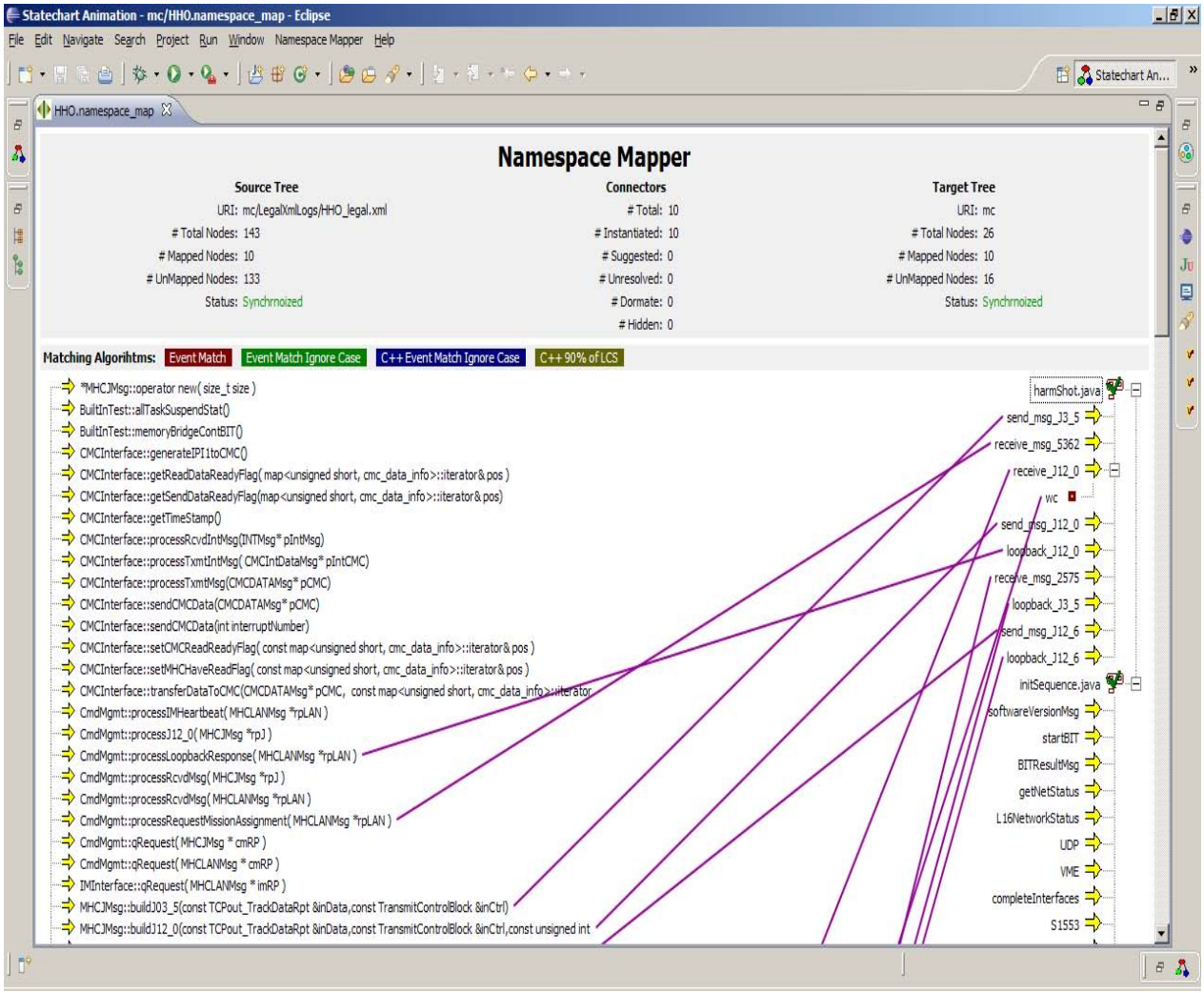


Figure 18. MC Namespace mapping.

8.5 Verification

The final step is of-course, verification. In this step the verification JUnit test (the JUnit equivalent of the log file) was executed in the assertion repository. It operates as follows. Every event in the verification Junit test (such as *assertions. fire("TerminalCtrl::snd1553_BIMA01()");* in Listing 4) corresponds to a method that fired and subsequently logged. This event is translated, using the executable namespace translation code, into the namespace of the assertion repository, and then dispatched to all assertions in the repository. Every assertion that contains that event responds by possibly changing states, while all others simply ignore it.

The assertion repository also keeps track and reports the following information, as depicted in Figure 19:

- Names of all assertions that failed (ended with *bSuccess*=false) during this verification JUnit test

- All SUT events the fired (and hence present in the log file and subsequently in the verification JUnit test) but no assertion with such an event name was detected. This situation could indicate an error in the namespace mapping.

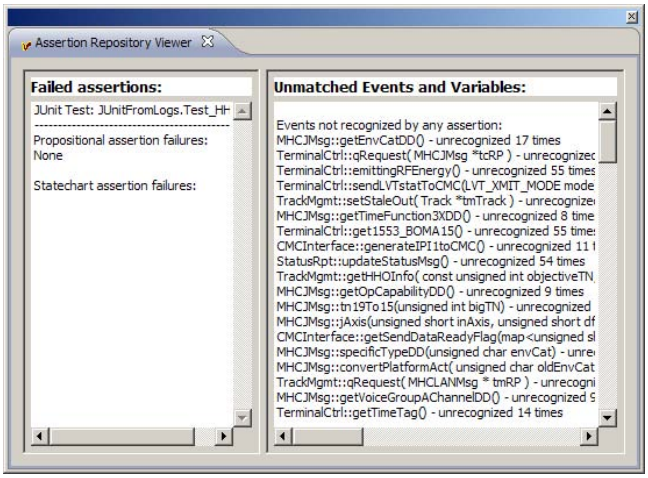


Figure 19. The assertion repository viewer showing all assertions that failed and all SUT events that fired but were not mapped to an assertion event.

8.6 Assertion Coverage Animation

We executed the verification JUnit test using the StateRover's animation using a special coverage animation option. The results of this execution, depicted in Fig. 20, color all states that were visited during the test as well as all transitions that were traversed during the test. This is done for all assertions of interest. Such coverage animation provides the following devil's advocate information: absence of animation coloring indicates interesting scenarios that could potentially cause an assertion to fail; such scenarios can then be coded as new on-target tests for the SUT.

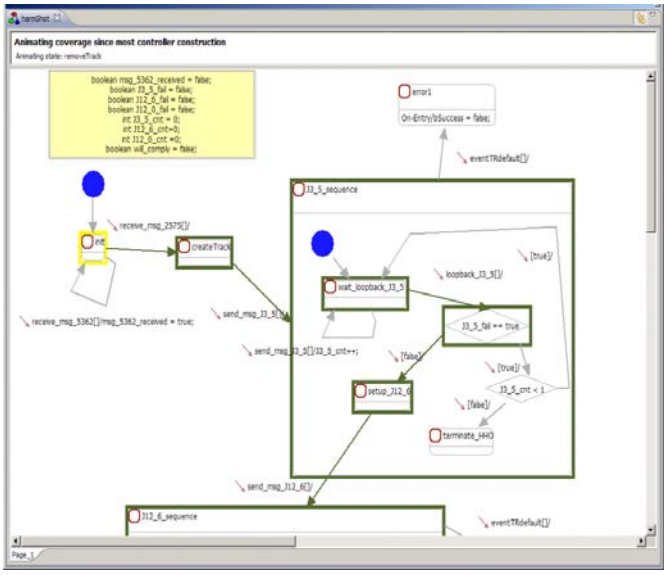


Figure 20 Coverage animation after the execution of a verification JUnit test

9. CONCLUSIONS

In this report, we presented an approach for formal validation and verification for mission- and safety-critical systems, and illustrated the approach with two case studies, a flight control system of the Brazilian Satellite Launcher and controller of the US Multifunctional Information Distribution System (MIDS), where UML statechart assertions are used to formally capture the complex mission-critical behaviors of these systems. The assertions are first validated via scenario-based testing and then used to automatically instrument the target software. The instrumented software are executed on the VxWorks-based target to generate log files, which are then imported into the StateRover SV&V environment and executed as JUnit tests against the assertions. This is the first reported cases of the application of the entire SV&V methodology using rapidly developed UML-based formal specifications. It is also the first reported case of "distributed verification", where specification and validation of assertions for the Brazilian Satellite Launcher was performed in California, execution and log file creation were performed in Brazil, and subsequent log-file based RV was performed in California. The StateRover SV&V tool supports all the activities involved in this approach, from the statechart assertions creation to the runtime verification against real data log files.

THIS PAGE INTENTIONALLY LEFT BLANK

REFERENCES

[1] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, “Experiences Using Lightweight Formal Methods for Requirements Modeling,” IEEE Trans. Software Eng., vol. 24, no. 1, pp. 4-14, Jan. 1998.

[2] D. Drusinsky and M. Shing, “Verification of Timing Properties in Rapid System Prototyping”, Proc.14th IEEE International Workshop in Rapid Systems Prototyping, 9-11 June 2003, pp. 47-53.

[3] J.B. Michael, D. Drusinsky, T. Otani and M. Shing, “Application of UML Statechart-based Verification and Validation for Trustworthy Software Systems”, manuscript, Sept. 2010.

[4] D. Drusinsky, M. Shing and K. Demir, “Creating and Validating Embedded Assertion Statecharts,” *IEEE Distributed Systems Online*, vol. 8, no. 5, 2007, art. no. 0705-o5003.

[5] http://en.wikipedia.org/wiki/VLS-1#References.

[6] D. Drusinsky, Modeling and Verification Using UML Statecharts – A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking, Elsevier, 2006.

[7] LEITE FILHO, W. C. . Control System of Brazilian Launcher. In: 4th ESA International Conference on Spacecraft Guidance, Navigation and Control Systems, 1999, Noordwijk, The Netherlands. Proc. of 4th ESA Inter. Conf. on GNC, 1999. p. 401-405.

[8] K. Beck and E. Gamma, “Test infected: Programmers love writing tests”, Java Report, 3(7), pp. 37-50, 1998.

[9] LEITE FILHO, W. C. ; CARRIJO, D. S. . Hardware in the Loop Simulation of Brazilian Launcher VLS. In: 3rd ESA International Conference on Spacecraft Guidance, Navigation and Control Systems, 1996, Noordwijk. Proc. of 3rd ESA Inter. Conf. on GNC, 1996. p. 355-358.

[10] LEITE FILHO, W. C. ; CARRIJO, D. S. ; OLIVA, A. P. . Hybrid Simulation Software Development for Assessment of a Satellite Launcher Control System. In: IASTED International Conference on Modelling and Simulation, 1998, Pittsburgh. Proc. Inter. Conf. on Modelling and Simulation, 1998. p. 245-249.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Professor Peter Denning
   Naval Postgraduate School
   Monterey, California

5. Professor Doron Drusinsky
   Naval Postgraduate School
   Monterey, California

6. Professor Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

7. Mr. Konstantin Beylin
   Naval Air Warfare Center, Weapons Division
   Point Mugu, CA 93042, USA

8. Dr. Miriam C. B. Alves
   Institute of Aeronautics and Space - Brazil
   c/o Naval Postgraduate School
   Monterey, California